Universidad de Buenos Aires

Facultad de Ingeniería

75.29
Teoría de Algoritmos
Trabajo Práctico Nº 1

Alumnos:

- Devesa, Leandro - 95637
- Rogica, Nicolás - 94107

Parte 1: Variante de Gale Shapley

Ejecución

Para correr el programa hay que tener instalado python.
El programa recibe 5 parámetros:

- Generar Archivos de datos nuevos: Boolean (0, 1)
- N, cantidad de recitales (1..n)
- M, cantidad de bandas: Int (1..n)
- X, máximo de bandas que pueden tocar en un recital: Int (1..n)
- Y, máximo de recitales que puede tocar una banda: Int (1..n)

OJO: A menos que se quiera hacer una prueba específica con datos anteriores (y por ende, mismos parámetros), siempre generar archivos de datos nuevos

Ejemplo de ejecución:

python varianteGaleShapley.py 1 10 10 1 1

Explicación y pseudocódigo del algoritmo

La implementacion se hizo en base a la variante “Hospitals/Residents” de Gale-Shapley (https://en.wikipedia.org/wiki/National_Resident_Matching_Program#Matching_algorithm)

El programa consiste en recorrer la lista de “Recitales” (cant n) e ir asignando la cantidad de “Bandas” (cant m) hasta que se llene el recital (x). Cada banda no puede tocar en mas de “y” recitales. Si al recorrer los recitales, se encuentra que una banda ya no puede tocar en un recital, entonces se analiza el orden de preferencia de la banda, y en caso de que esta prefiera este otro recital entonces se la desasigna del recital anterior para asignarlo en el correspondiente.

```
While #Recitales > 0
        While BandasPreferidasEnRecital
                If CantBandasXRecital < x
                        If BandaYaTocaEnOtroRecital
                                If CantRecitalesXBanda < y
                                        agregarBandaARecital
                                else
                                        analizarQueRecitalPrefiereLaBanda
                                        agregarRecitalQueSeLibero
                                end if
                        else
                                agregarBandaARecital
```

```
                    end if
            end while
end while
```

Complejidad del algoritmo

Debido a los ciclos while anidados, la complejidad algorítmica es de O(log n)

Condiciones para matching estable y perfecto

Para que el matching sea estable la cantidad de bandas debe ser igual a la cantidad de recitales y además la cantidad de bandas por recital (x) y la cantidad de recitales por banda (y) deben ser iguales . Esto garantiza el mejor match basado en las preferencias de cada uno (matching perfecto).
En el caso que la cantidad de bandas o de recitales sean distintos para generar un matching estable debe cumplirse que la cantidad de recitales sea el doble que la cantidad de bandas y que la cantidad de bandas por recital (x) sea igual que la cantidad de recitales por banda(y)

¿Qué sucede si hay preferencias similares?
Si hay preferencias similares, la complejidad se acerca lo más posible a la cota O() (es el peor caso)

En el caso que se decida desempatar tirando una moneda esto provocaría que no se pueda conseguir un matching perfecto, la inclusión de está operación no afectaría la complejidad del algoritmo.

Variaciones al modificar los parámetros de entrada

N = 10 M = 10 X = 1 Y = 1 genera un matching estable y perfecto
N = 10 M = 5 X = 2 Y = 2 genera un matching estable y perfecto
N = 10 M = 5 X = 2 Y = 1 genera matching estable pero no perfecto, hay bandas que quedan sin asignar y los recitales quedan incompletos

Parte 2: Complejidad algorítmica

Ejecución

El programa recibe 2 parámetros:

- N, número hasta el cual se quiere calcular los primos
- Modo de ejecución, 'F' para calcular por fuerza bruta, 'E' para calcular por el método de Eratóstenes

Ejemplo de ejecución:

python criba.py 100 F

Pseudocódigo del algoritmo para calcular números primos menores a N

tomar todos los números hasta N, asumir que son primos y almacenarlos en un array K
para cada número n entre 2 y N :
    si Kn es primo:
        indicar que n es el factor primo de valor mínimo para Kn
        almacenarlo en un array P
    indicar que n es el factor primo de valor mínimo para todos los números Ki donde i < |P|, Pi <= Kn, n * Pi <= N, Ki = n * Pi

Complejidad del algoritmo

Dado un conjunto N de números almacenados en una estructura la primer parte del algoritmo establece que cada elemento va a ser procesado al menos una vez, pero además la segunda parte establece que van a ser procesados sólo una vez. Esto se debe a que la primer condición solo permite que se accedan a los valores Ki que se pueden calcular como resultado de la operación "a*b" dónde a es el factor primo de menor valor para Ki y b no puede tener factores primos de menor valor que a (ya que todavía no fueron encontrados). Esto hace que la complejidad del algoritmo sea O(n)

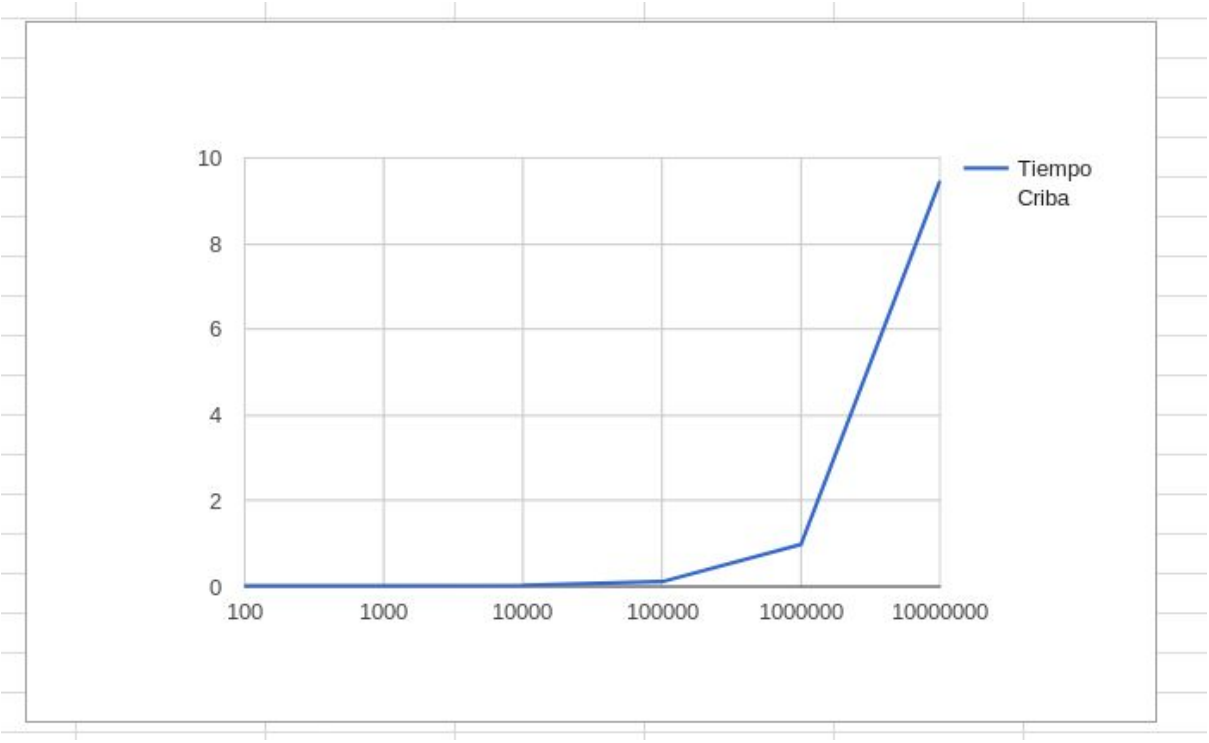
Algoritmo de fuerza bruta para calcular números primos

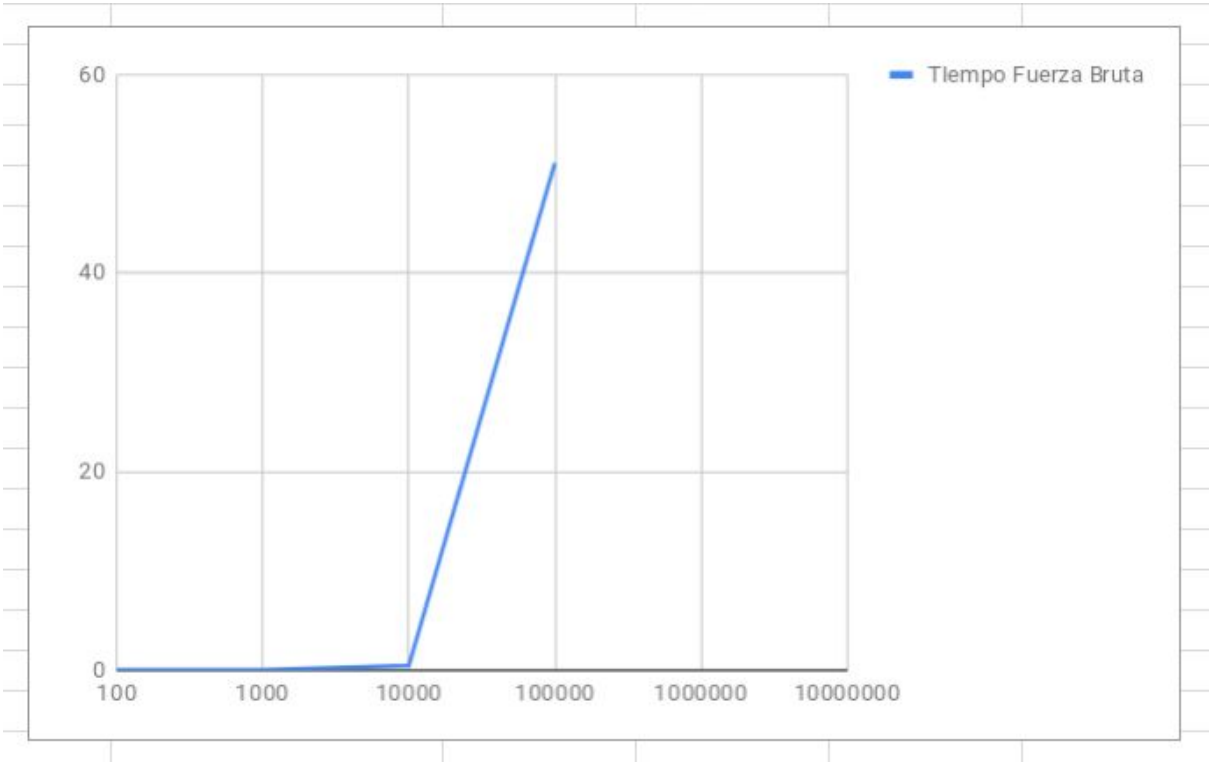
```
para cada número n entre 2 y N:
    esPrimo ← true
    para número j entre 2 y n-1:
        si n mod j = 0:
            esPrimo ← false
            break
```

```
if esPrimo = true:
    almacenar n en un array
```

Al igual que el algoritmo anterior dado un conjunto de N números en este también se van a procesar todos pero además cuenta con un segundo ciclo en el cual para el peor caso dado un número n se van a realizar n-1 operaciones sobre el mismo por lo que la complejidad es $O(n^2)$

Gráficos para los tiempos de ejecución





Se puede observar como ambos algoritmos mantienen cierto nivel de paridad en los valores iniciales 100,1000,10000, momento a partir del que si bien empieza a crecer el tiempo de ejecución en el algoritmo por fuerza bruta este crecimiento es muy abrupto, mientras que la criba en el orden de segundos, el algoritmo por fuerza bruta termina aumentando su tiempo de ejecución en minutos y horas (en este caso no está graficado pero el tiempo de ejecución para N= 1000000 y N= 10000000 supera la hora en el primer caso y las dos horas en el segundo).

Aclaración: los tiempos de ejecución se calcularon usando el comando time de linux y omitiendo la salida por pantalla de la lista de números primos, se tomó como referencia el tiempo de usuario.